

CS190C Lec5

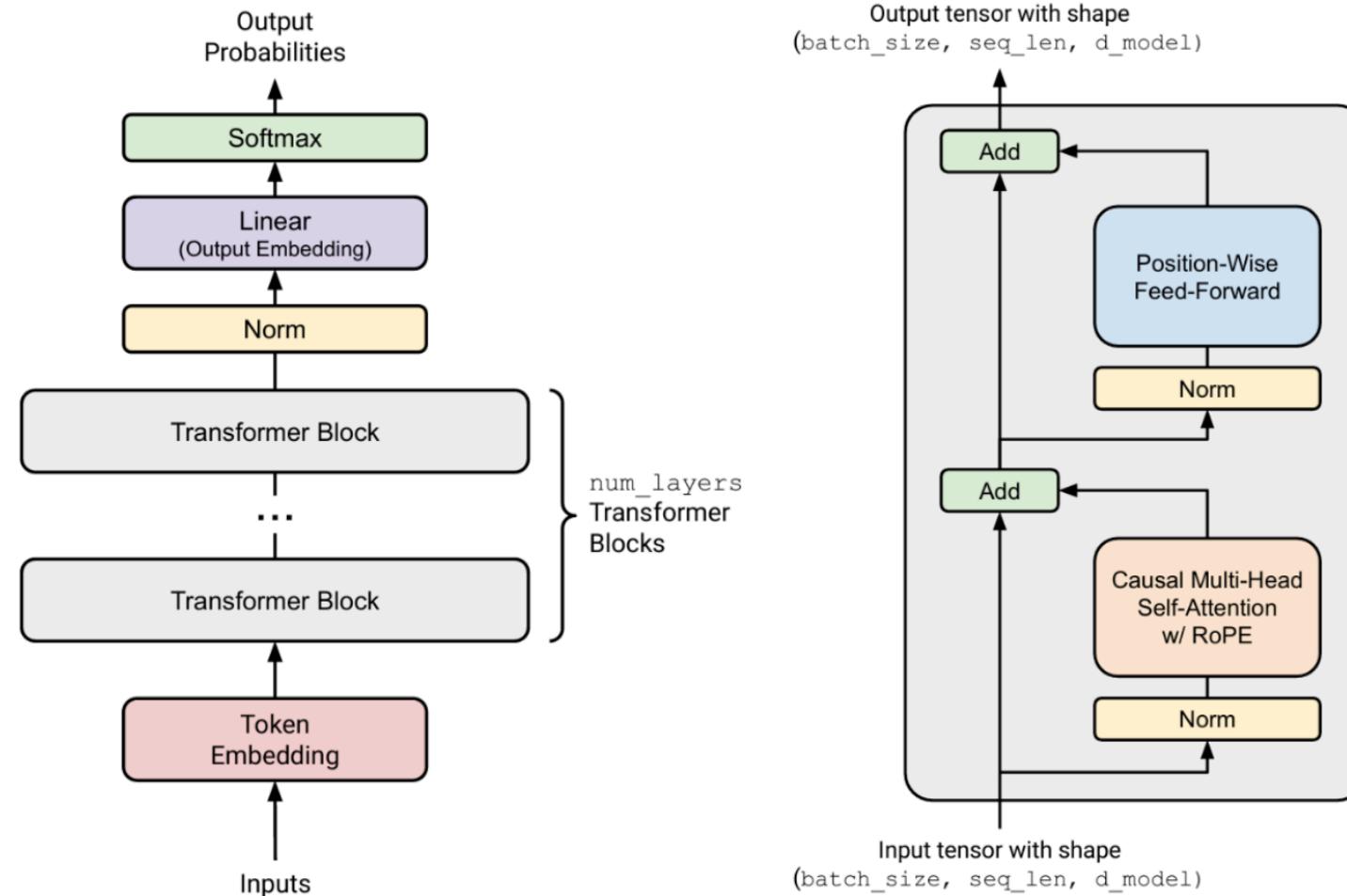
Build Transformer Decoder

Overview

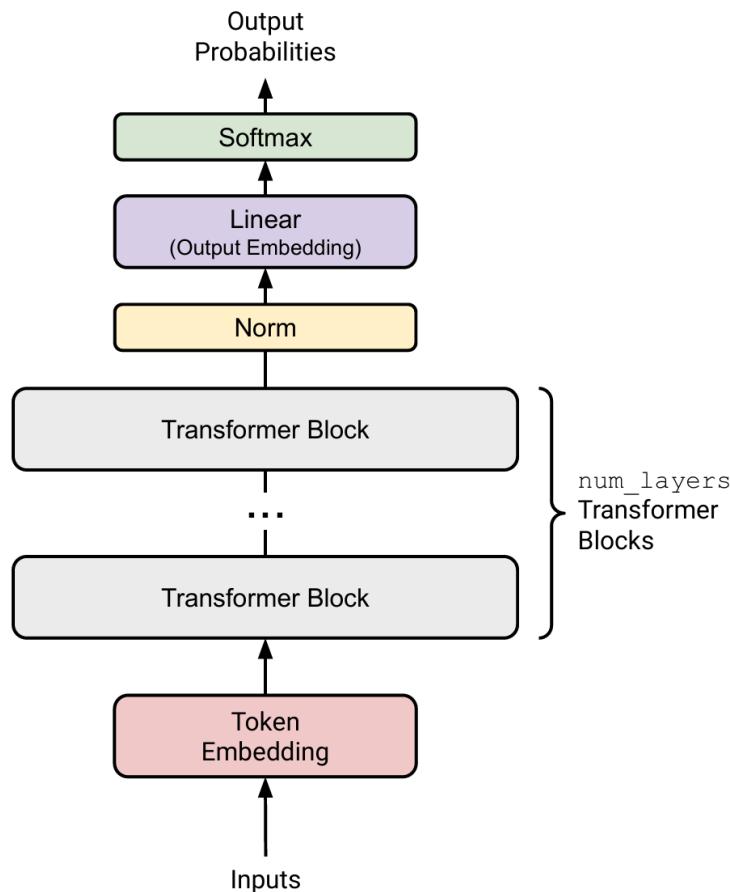
- Macro System Architecture
- Code Implementation of Basic Computational Components
- Code Implementation of RoPE
- Code Implementation of FFN and Attention
- Assembly of Transformer Block and Final Model

PART1: Macro System Architecture

Transformer Overall Architecture Diagram

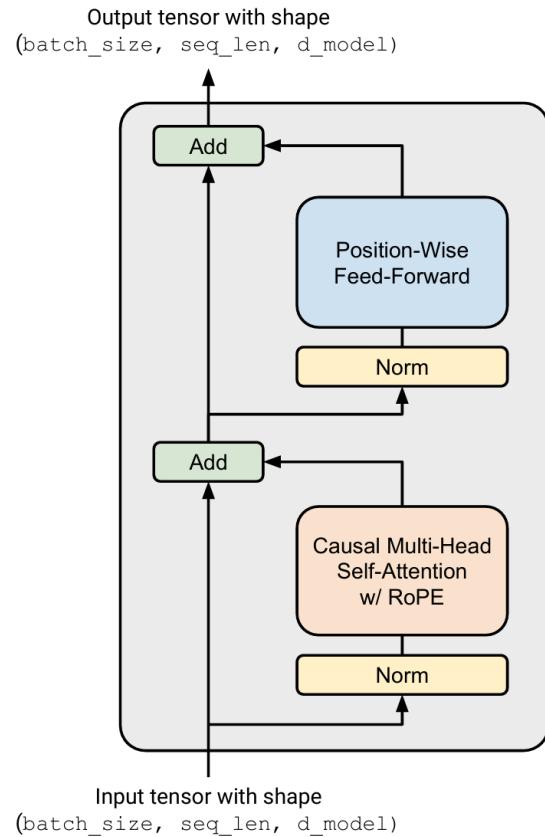


Modules of the whole Transformer



- Pass through a **Token Embedding** module to turn to dense vector (Input layer)
- Pass through several **Transformer Block** modules to absorb information in multiple rounds (Hidden layer)
- Normalize the number scale of final tensor
- Linear transform, calculate possible scores for generating of each word (Output layer)

Modules of each Transformer Block

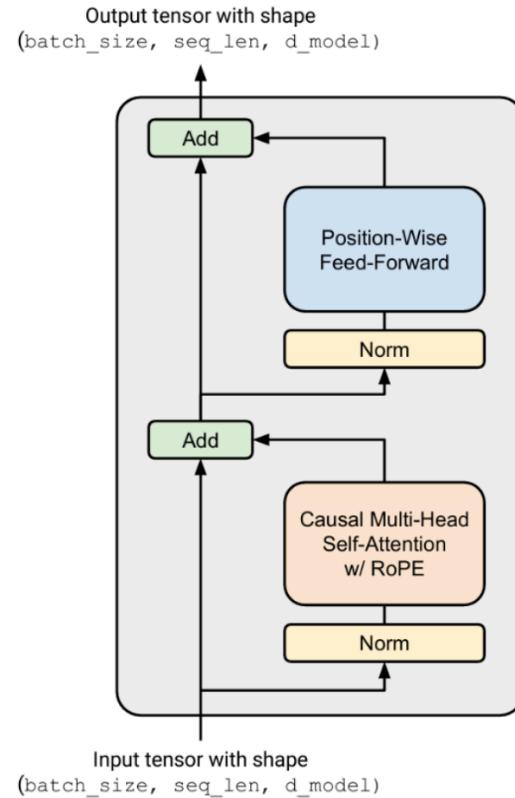
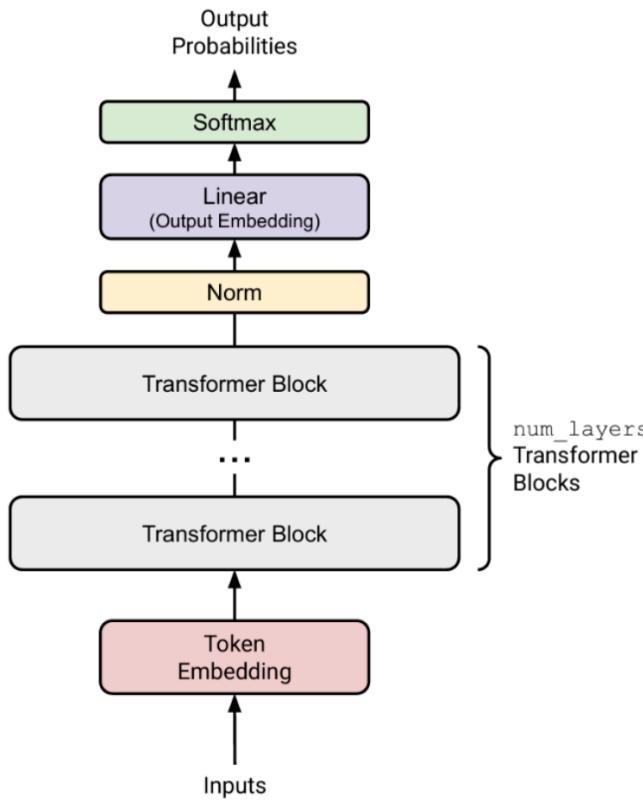


- Pre-layer RMSNorm module
- MHA Module with Residual Connection
 - contains RoPE module if position embedding needed
 - contains Softmax module for calculation of attention score
- Pre-layer RMSNorm module
- FFN Module with Residual Connection
 - contains SiLU module

Moreover, almost all modules require calling the **linear transformation** module!

PART2: Code Implementation of Basic Computational Components

Basic Components to be Implemented



Modules to be implemented:

- Token Embedding
- Linear
- RMSNorm
- SiLU
- Softmax

1. class Generate_EMBEDDINGS

IDEA:

- Most original input: BPE encoding results (e.g., [3,10,2,6,4], all token IDs from the vocabulary)
- Tensor shape: [batch_size, seq_len]
- Expected model input: Different words in the vocabulary have different embedding vectors $\text{emb}_i \in R^{d_model}$
- Implementation idea:
 - Generate a learnable matrix $W_e \in R^{|V|*d_model}$.
 - The i -th column represents emb_i
 - Randomly initialized W_e , representing no prior knowledge about the meaning of any word at first.
 - Learn W_e with other parameters during the training process

1. class Generate_EMBEDDINGS

Parameter scheme:

- Initialization phase: Pass in `vocab_size`, `d_model`, `device` (the device where PyTorch tensors are stored), and `dtype` (numerical type of tensor elements).
- Forward phase:
 - Pass in `token_ids` (shape `[batch_size, seq_len]`)
 - Output shape is `[batch_size, seq_len, d_model]`

1. class Generate_EMBEDDINGS

```
# vocab_size*embedding_dim矩阵中取样
class Generate_EMBEDDINGS(nn.Module):

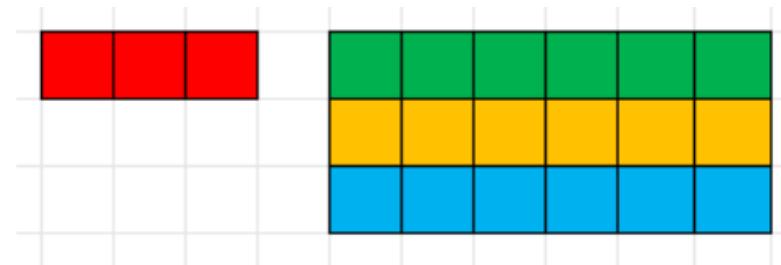
    def __init__(self,number_embeddings:int,embedding_dim:int,device=None,dtype=None):
        super(Generate_EMBEDDINGS,self).__init__()
        self.embedding_matrix=torch.empty(number_embeddings,
                                         embedding_dim,
                                         device=device,
                                         dtype=torch.float32)
        nn.init.trunc_normal_(self.embedding_matrix,mean=0,std=0.02)

    def forward(self,token_ids:torch.Tensor)->torch.Tensor:
        return self.embedding_matrix[token_ids]
```

2. class Linear_Transform

IDEA:

- Assume we need to transform a 3-dimensional tensor into a 6-dimensional one...
- Mathematically speaking: Let the 3D tensor x ($1 * 3$) right-multiply a $3 * 6$ matrix w
- The shape of xw is then $1 * 6$, just like the diagram below.

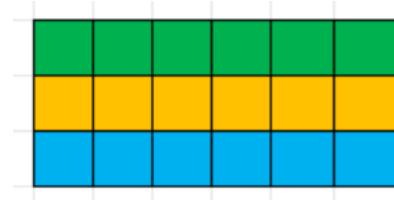


Question: Linear operations are the most frequent in LLMs.....

Can this operation be accelerated as much as possible?

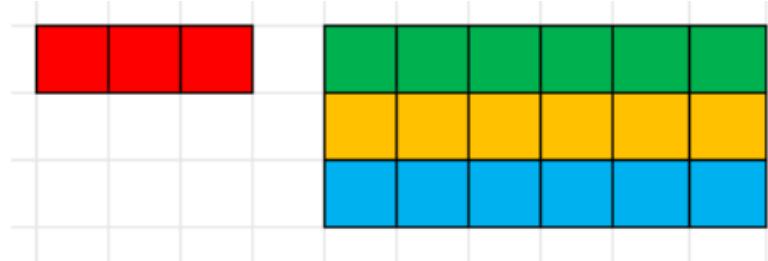
2. class Linear_Transform

- PyTorch tensors follow the "last dimension elements have contiguous memory addresses" principle.
- For a $3 * 6$ tensor W, its memory layout as the diagram(same color means contiguous):



- For a $4 * 3 * 6$ tensor ([batch size, rows, columns]), every 6 elements are also contiguous (e.g., the memory address difference between [1,1,1] and [1,1,2] is 1 units, and the memory address difference between [1,1,1] and [1,2,1] is 6 units).
- That is: Matrix elements in the same row have contiguous memory, while those in

2. class Linear_Transform

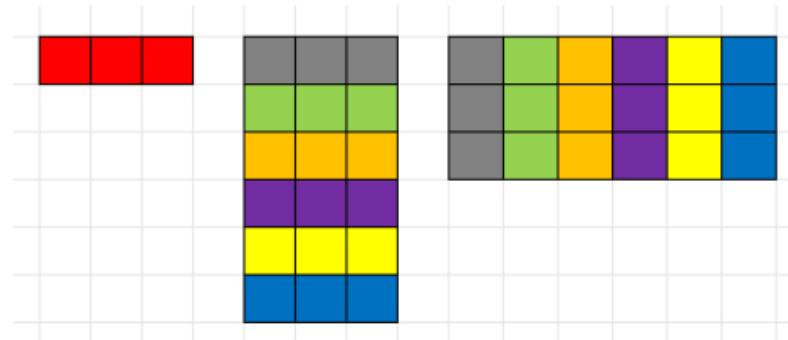


Perform 6 vector dot product operations. In each operation:

- x participates as a whole row, memory is contiguous, can fully utilize cache
- w participates as a whole column, memory is not contiguous, may not utilize cache effectively

How can we make w also participate "as a whole row" in each operation?

2. class Linear_Transform



- PyTorch's transpose operation does not change the tensor's underlying memory space, just change stepsize. (Can you give an example?)
- Create new `w : [6, 3]` (every 3 elements are contiguous in memory)
- Transpose `w` to `[3, 6]` : Can perform matrix multiplication, and the memory distribution remains unchanged
- `x` and `w` have fully contiguous memory access during each operation, allowing full utilization of cache!

2. class Linear_Transform

```
# 重点：行主序规则
class Linear_Transform(nn.Module):

    def __init__(self,in_features:int,out_features:int,device=None,dtype=None):
        super(Linear_Transform,self).__init__()
        self.linear_matrix=torch.empty(out_features,
                                      in_features,
                                      device=device,
                                      dtype=torch.float32)

        self.linear_matrix=self.linear_matrix.transpose(-2,-1)
        nn.init.trunc_normal_(self.linear_matrix,mean=0,std=0.02)
        self.linear_matrix=nn.Parameter(self.linear_matrix)

    def forward(self,x:torch.Tensor)->torch.Tensor:
        return torch.matmul(x,self.linear_matrix)
```

3. class RMSNorm

IDEA: Normalize the input tensor \vec{a} (We've discussed why at Lec3)

- $a_i = \frac{a_i}{RMS(\vec{a})} g_i$ (divide by normalization weight uniformly, and apply learnable fine-tuning)
- g_i is a learnable parameter
- $RMS(\vec{a}) = \sqrt{\left(\frac{1}{d_{model}} \sum a_i^2\right) + \epsilon}$, that is L2-Norm.

Input tensor shape: [batch_size, seq_len, d_model] \Rightarrow Not a 1D vector, how to handle?

PyTorch's broadcasting mechanism:

- Operations are performed on the last few dimensions by default, previous dimensions are all replication operations
- Equivalent to parallel operations on batch_size * seq_len parallel d_model -

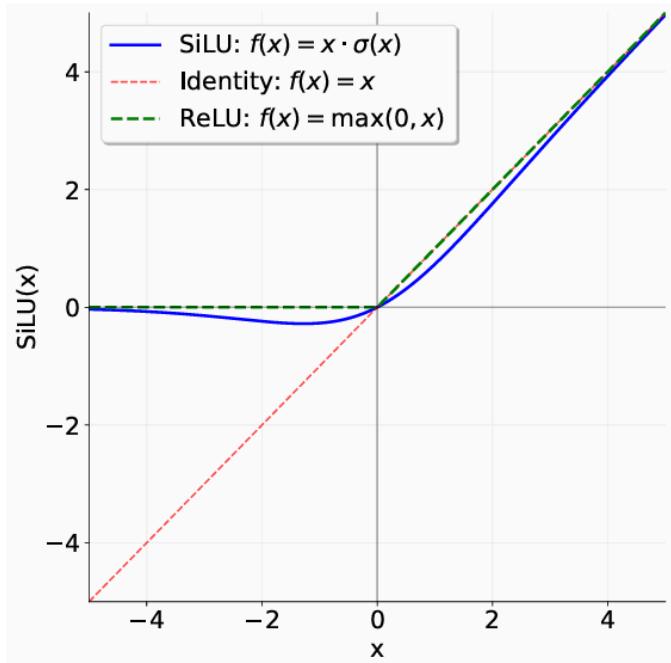
3. class RMSNorm

```
# 运算法则：归一化*可学习缩放倍数
class RMSNorm(nn.Module):
    def __init__(self,d_model:int,eps:float=1e-5,device=None,dtype=None):
        super(RMSNorm,self).__init__()
        self.eps=eps
        self.g=nn.Parameter(torch.ones(d_model,device=device,dtype=torch.float32))

    def _get_rms(self,x:torch.Tensor)->torch.Tensor:
        sum_square=torch.sum(x**2,dim=-1,keepdim=True)
        mean_square=sum_square/x.shape[-1]
        return torch.sqrt(mean_square+self.eps)

    def forward(self,x:torch.Tensor)->torch.Tensor:
        ori_dtype=x.dtype
        x=x.to(torch.float32) # 张量元素类型转换，确保运算精度
        rms=self._get_rms(x)
        x_normed=x/rms
        x_normed=x_normed.to(ori_dtype) # 张量元素类型还原
        return x_normed*self.g
```

4. class SiLU_Activation



$$\text{SiLU}: f(x) = x \cdot \sigma(x)$$

$$\text{ReLU}: f(x) = \max(0, x)$$

- For $x < 0$: roughly equals to 0
- For $x > 0$: roughly stays the same
- Compare with ReLU: smooth and differentiable at $x = 0$

4. class SiLU_Activation

```
class Sigmoid_Activation(nn.Module):
    def __init__(self):
        super(Sigmoid_Activation, self).__init__()

    def forward(self, x:torch.Tensor)->torch.Tensor:
        denominator=1+torch.exp(-x)
        return 1/denominator

class SiLU_Activation(nn.Module):
    def __init__(self):
        super(SiLU_Activation, self).__init__()
        self.sigmoid_activator=Sigmoid_Activation()

    def forward(self, x:torch.Tensor)->torch.Tensor:
        sigmoid_x=self.sigmoid_activator(x)
        return x*sigmoid_x
```

5. class Softmax_Activation

How to turn a score tensor to a distribution?

$$x_i = \frac{e^{x_i}}{\sum e^{x_i}}$$

- Each x_i calculates its exponential as a weight, then performs weight normalization
- Can make the advantage of relatively larger values more pronounced
- Even smaller values remain non-zero after normalization

Problem: What if there exists a very large x_i ? (e.g., normalizing [20, 3, 1005])

- $e^{1000} = \text{NAN}$

5. class Softmax_Activation

- Normalizing $[100, 101, 102]$ vs Normalizing $[-2, -1, 0]$
- Weight of 102: $\frac{e^{102}}{e^{102} + e^{101} + e^{100}} = \frac{e^0}{e^0 + e^{-1} + e^{-2}}$
- The Softmax normalization result of $[100, 101, 102]$ is equivalent to that of $[-2, -1, 0]$

Let x_{max} be the maximum value among x_i :

$$\begin{aligned}\text{Softmax}(x_i) &= \frac{e^{x_i}}{\sum e^{x_i}} \\ &= \frac{e^{x_i}/e^{x_{max}}}{\sum e^{x_i}/e^{x_{max}}} \\ &= \frac{e^{x_i-x_{max}}}{\sum e^{x_i-x_{max}}}\end{aligned}$$

That is: subtract x_{max} from all x_i to avoid problems with extremely large values that cannot be calculated!

5. class Softmax_Activation

```
class Softmax_Activation(nn.Module):
    def __init__(self, dim:int=-1):
        super(Softmax_Activation, self).__init__()
        self.dim=dim

    def forward(self, x:torch.Tensor)->torch.Tensor:
        #shape of x:(bsz,seq_len,d_k)
        x_max=torch.max(x, dim=self.dim, keepdim=True).values
        x_exp=torch.exp(x-x_max)
        x_exp_sum=torch.sum(x_exp, dim=self.dim, keepdim=True)
        return x_exp/x_exp_sum
```

x_max : [bsz, seq_len] , x : [bsz,seq_len,d_k] .

x-x_max exhibits another type of broadcasting mechanism: replication operation

PART3: Code Implementation of RoPE

Review: RoPE Calculation Rules

$$\vec{x} \Rightarrow R_i \vec{x} \ (\vec{x} \in R^d, d \text{ is even})$$

Divide the d-dimensional vector into several sub-segments of length 2, resulting in a total of $d/2$ sub-segments, each sub-segment makes a rotation of angle $\theta_{i,k}$.
(Proportional to position i)

$$R^i = \begin{bmatrix} R_1^i & 0 & 0 & \cdots & 0 \\ 0 & R_2^i & 0 & \cdots & 0 \\ 0 & 0 & R_3^i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & R_{d/2}^i \end{bmatrix} \quad R_k^i = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}.$$

$$\text{where } \theta_{i,k} = \frac{i}{\Theta^{2k/d}}$$

Review: RoPE Calculation Rules

$$R^i = \begin{bmatrix} R_1^i & 0 & 0 & \cdots & 0 \\ 0 & R_2^i & 0 & \cdots & 0 \\ 0 & 0 & R_3^i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & R_{d/2}^i \end{bmatrix} \quad R_k^i = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}.$$

Property of matrix R : $(R^m)^T R^n = R^{n-m}$

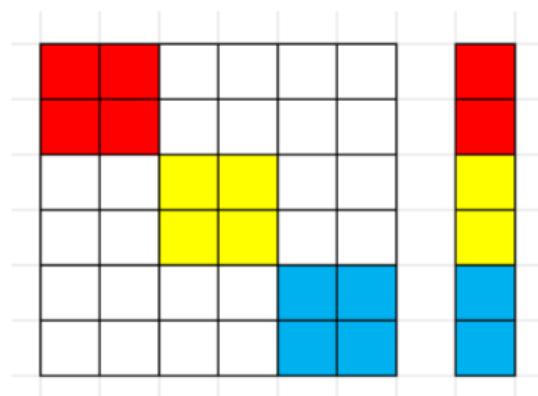
In the attention mechanism, for the query vector q_i at position i and the key vector k_j at position j :

- $q'_i = R^i q_i, k'_j = R^j k_j$
- $q_i'^T k'_j = q_i^T (R^i)^T R^j k_j = q_i^T R^{j-i} k_j$

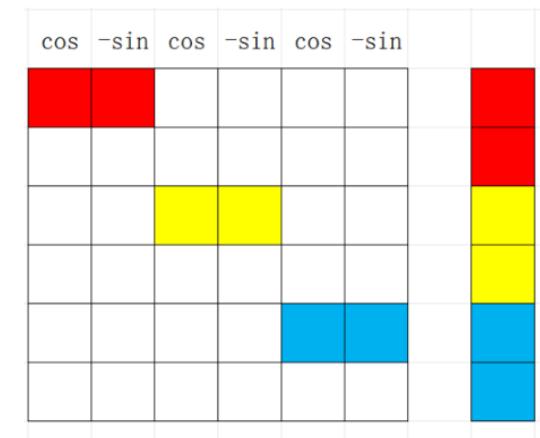
How to Avoid Brute-force Matrix Multiplication?

- Observation: Even rows of the matrix correspond to the same transformation rule:
 $\begin{bmatrix} \cos & -\sin \end{bmatrix}$, with angle $\theta_{i,k}$ as the variable
- If these even rows sharing the same rule can be computed efficiently in a unified manner, how should subsequent processing proceed?

$$R^i = \begin{bmatrix} R_1^i & 0 & 0 & \cdots & 0 \\ 0 & R_2^i & 0 & \cdots & 0 \\ 0 & 0 & R_3^i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & R_{d/2}^i \end{bmatrix}$$



$$R_k^i = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}.$$



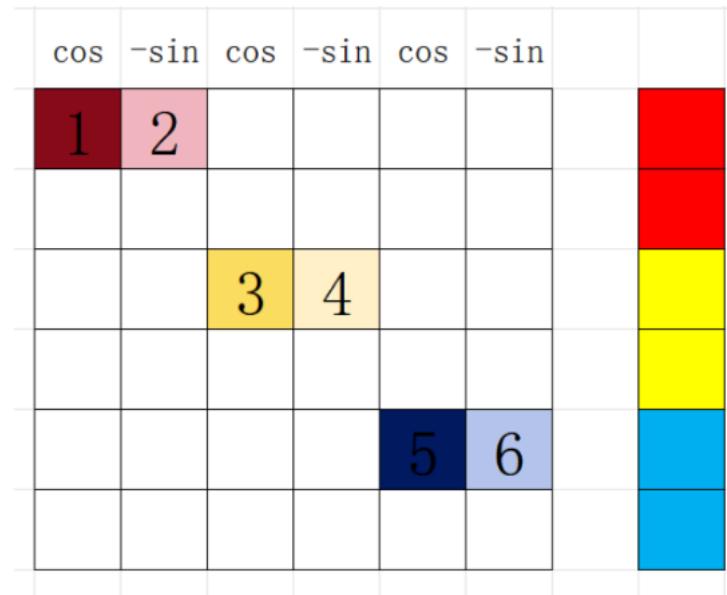
How to Avoid Brute-force Matrix Multiplication?

- Calculate products of Red-Red, Yellow-Yellow, Blue-Blue, which correspond to: x_0, x_2, x_4 after RoPE
- x_1, x_3, x_5 are the same.
- Calculate each 2-number blocks, and calculate products.



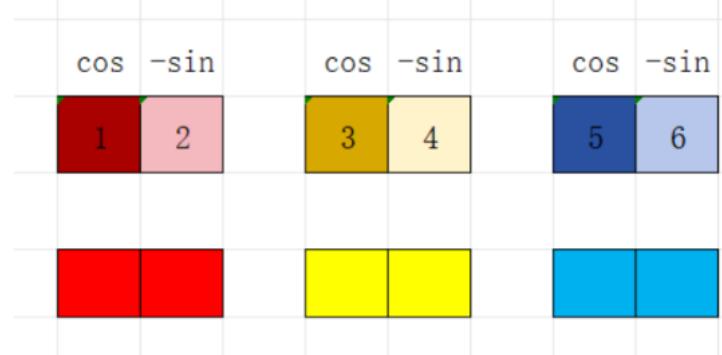
How to Calculate 2-number blocks of R_i ?

- Pre-calculate a $\theta_{i,k}$ lookup table of size [max_seq_len, d/2]
- Then calculate lookup tables of the same size for $\cos(\theta_{i,k})$ and $\sin(\theta_{i,k})$
- Obtain values at positions 1, 3, 5 directly from the cos lookup table
- Obtain values at positions 2, 4, 6 directly from the sin lookup table



How to Calculate 2-number blocks of R_i ?

- Concatenate pairs 12, 34, 56 into 3 blocks, perform block-wise dot product with the 3 blocks of x , obtaining the values for the three even rows.
- Similarly, obtain the values for all odd rows of the transformed x , concatenate both to get the complete transformed x vector.



Code Implementation of RoPE

```
class RoPE(nn.Module):
    def __init__(self, theta:float,d_k:int,max_seq_len:int,device=None):
        super(RoPE,self).__init__()
        self.theta=theta
        self.d_k=d_k
        self.max_seq_len=max_seq_len
        self.device=device

        d_half=d_k//2
        positions=torch.arange(max_seq_len, device=device).unsqueeze(1)#[max_seq_len,1]
        dims=torch.arange(d_half,device=device).unsqueeze(0)#[1,d_half]
        angles=positions/(theta**((2*dims/d_k)))#[max_seq_len,d_half]

        cos_values=torch.cos(angles).unsqueeze(0)
        self.register_buffer("cos_values",cos_values) #(1,max_seq_len,d_half)
        sin_values=torch.sin(angles).unsqueeze(0)
        self.register_buffer("sin_values",sin_values) #(1,max_seq_len,d_half)
```

max_seq_len : maximum iteration value of i ; angles : values of $\theta_{i,k}$

positions : sequence of i values; dims : sequence of k values

Code Implementation of RoPE

```
def forward(self,x:torch.Tensor,token_positions:torch.Tensor)->torch.Tensor:  
    x_spilited=x.reshape(*x.shape[:-1],self.d_k//2,2)  
    cos_chunk=self.cos_values[:,token_positions,:]  
    sin_chunk=self.sin_values[:,token_positions,:]  
  
    even_transform=torch.stack([cos_chunk,-sin_chunk],dim=-1)  
    odd_transform=torch.stack([sin_chunk,cos_chunk],dim=-1)  
  
    x_rotated_odd=torch.sum(x_spilited*even_transform, dim=-1)#(bsz,seq_len,d_k//2)  
    x_rotated_even=torch.sum(x_spilited*odd_transform, dim=-1)#(bsz,seq_len,d_k//2)  
    stacked_x=torch.stack([x_rotated_odd,x_rotated_even], dim=-1)  
    x_rotated=stacked_x.reshape(*stacked_x.shape[:-2],self.d_k)  
  
    return x_rotated
```

cos_chunk : d_half dimensional lookup table for each i position (positions 1, 3, 5)

PART4: Code Implementation of FFN and Attention

1. class Feed_Forward_Network

Module algorithm (excluding residual connection):

- Input tensor x (d_{model} dimensional)
- Route 1: Transform via w_1 to d_{ff} dimensional, pass through $SiLU$ activation function to get new x
- Route 2: Transform via w_3 for another expansion, getting gated d_{ff} dimensional
- Element-wise multiplication of x and gated, getting new x
- Transform back to d_{model} dimensional via w_2

We call it $\text{SwiGLU} : \text{FFN}(x) = \text{SwiGLU}(x, W_1, W_2, W_3)$

1. class Feed_Forward_Network

```
class Feed_Forward_Network(nn.Module):
    def __init__(self,
                 d_model:int,
                 d_ff=None,
                 device=None,
                 dtype=None):
        super(Feed_Forward_Network,self).__init__()

        self.d_model=d_model
        if d_ff is not None:
            self.d_ff=d_ff
        else:
            self.d_ff=int(8/3*d_model)

        self.linear_w1=Linear_Transform(d_model,d_ff,device=device,dtype=dtype)
        self.linear_w3=Linear_Transform(d_model,d_ff,device=device,dtype=dtype)
        self.linear_w2=Linear_Transform(d_ff,d_model,device=device,dtype=dtype)
        self.activator=SiLU_Activation()
```

1. class Feed_Forward_Network

```
def forward(self,x:torch.Tensor)->torch.Tensor:  
    enhanced=self.linear_w1(x)  
    activated=self.activator(enhanced)  
    gate=self.linear_w3(x)  
    gated=activated*gate  
    output=self.linear_w2(gated)  
    return output
```

2. `class Multihead_Attention`

Module algorithm (excluding residual connection):

- Input tensor `x` with size `[batch_size, seq_len, d_model]`
- Through three linear transformations, essentially linearly transforming each `d_model` dimensional word vector of `x` to get:
 - `Q`, `K` matrices `[batch_size, seq_len, num_heads*d_k]`
 - `V` matrix `[batch_size, seq_len, num_heads*d_v]`
- Perform `RoPE` positional encoding on `Q`, `K` matrices
- Generate attention upper triangular mask (to prevent breaking the "autoregressive" assumption)
- Use QKV matrices and attention mask to compute attention output

2. class Multihead_Attention

Method for calculating attention output:

- Multiply Q , K matrices to calculate token feature matching scores
- Apply mask processing to the matching score matrix
- $\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$, calculate attention scores between tokens
- Multiply the score matrix by the v matrix to get the attention output of each head
- Multiply w_0 matrix to integrate heads

Calculation of Attention Output for One Head

```
class Scaled_dot_Product_Attention(nn.Module):
    def __init__(self):
        super(Scaled_dot_Product_Attention, self).__init__()

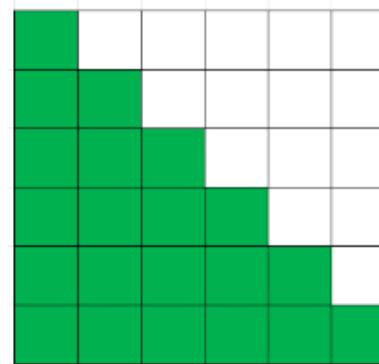
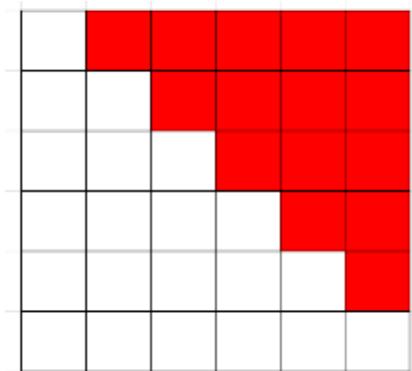
    def forward(self,Q:torch.Tensor,K:torch.Tensor,V:torch.Tensor,mask:torch.Tensor=None)->torch.Tensor:
        d_k=Q.shape[-1]

        attn_score=torch.matmul(Q,K.transpose(-2,-1))/math.sqrt(d_k)#[bsz,*,Qseq_len,Kseq_len]
        if mask is not None:
            attn_score=attn_score.masked_fill(mask==0,float('-inf'))
        softmax=Softmax_Activation(dim=-1)
        attn_weight=softmax(attn_score)#[bsz,*,Qseq_len,Kseq_len]

        #V:[bsz,*,Kseq_len,d_v]
        attn_output=torch.matmul(attn_weight,V)
        return attn_output#[bsz,*,Qseq_len,d_v]
```

Generation of Attention Mask

```
class Causal_Mask:  
    def __init__(self,seq_len,device=None):  
        self.seq_len=seq_len  
        self.device=device  
  
    def generate(self)->torch.Tensor:  
        ones=torch.ones(self.seq_len,self.seq_len,device=self.device)  
        mask=torch.triu(ones,diagonal=1)  
        mask=(mask==0)  
        return mask
```



List of Sub-modules in Multihead_Attention

- Attention mask generation module
- Attention output calculation module (sdpa)
- RoPE module ⇒ Requires additional parameters like max_seq_len, theta, token_positions, etc.
- Four types of linear transformations: Q, K, V, O

Assembly of Complete Module

```
class Multihead_Attention(nn.Module):
    def __init__(self,
                 d_model:int,
                 num_heads:int,
                 max_seq_length:int=None,
                 theta:int=None,
                 device=None):
        super(Multihead_Attention, self).__init__()

        self.d_model=d_model
        self.num_heads=num_heads
        self.d_k=d_model//num_heads
        self.d_v=d_model//num_heads

        self.max_seq_length=max_seq_length
        self.theta=theta
        self.token_positions=None

        self.q_proj=Linear_Transform(d_model,num_heads*self.d_k,device=device)
        self.k_proj=Linear_Transform(d_model,num_heads*self.d_k,device=device)
        self.v_proj=Linear_Transform(d_model,num_heads*self.d_v,device=device)
        self.o_proj=Linear_Transform(num_heads*self.d_v,d_model,device=device)

        self.sdpa=Scaled_dot_Product_Attention()

    if max_seq_length is not None and theta is not None:
        self.rope=RoPE(theta,self.d_k,max_seq_length,device=device)
    else:
        self.rope=None
```

Assembly of Complete Module

```
def forward(self,x:torch.Tensor,token_positions:torch.Tensor=None)->torch.Tensor:  
    bsz=x.shape[0]  
    seq_len=x.shape[1]  
  
    #qk:[bsz,n_heads,seq_len,d_k]  
    #v:[bsz,n_heads,seq_len,d_v]  
    Q=self.q_proj(x)  
    Q=Q.reshape(bsz,seq_len,self.num_heads,self.d_k).transpose(1,2)  
    K=self.k_proj(x)  
    K=K.reshape(bsz,seq_len,self.num_heads,self.d_k).transpose(1,2)  
    V=self.v_proj(x)  
    V=V.reshape(bsz,seq_len,self.num_heads,self.d_v).transpose(1,2)  
  
    #apply RoPE on QK  
    if self.rope is not None:  
        self.token_positions=token_positions  
        Q=self.rope(Q,self.token_positions)  
        K=self.rope(K,self.token_positions)
```

Assembly of Complete Module

```
mask=Causal_Mask(seq_len,device=x.device).generate()  
mask=mask.unsqueeze(0).unsqueeze(1)  
  
attn_output=self.sdpa(Q,K,V,mask)  
attn_output=attn_output.transpose(1,2).reshape(bsz,seq_len,self.num_heads*self.d_v)  
attn_output=self.o_proj(attn_output)  
  
return attn_output
```

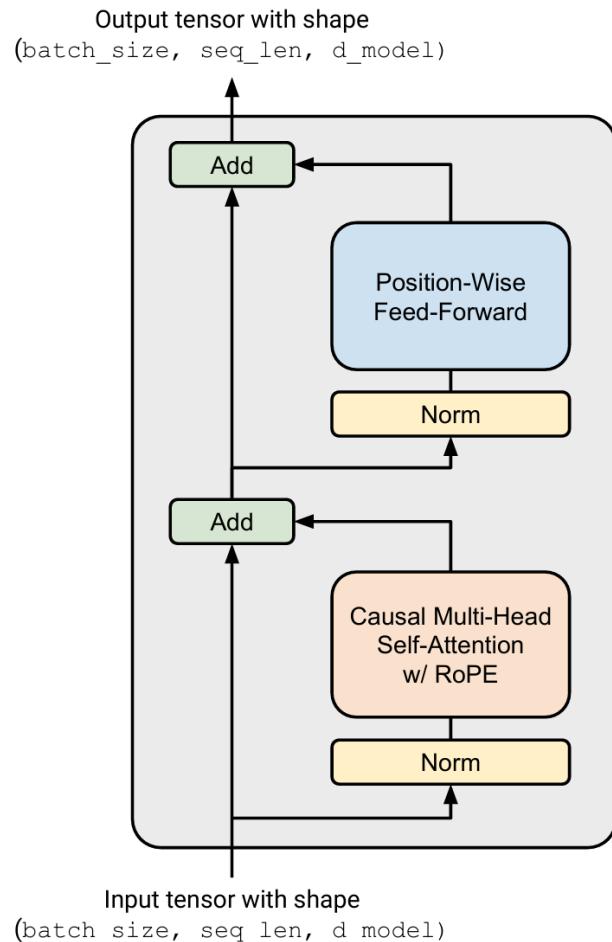
Original attn_output: [bsz, num_heads, seq_len, d_v]

[bsz, seq_len, num_heads, d_v] \Rightarrow [bsz, seq_len, num_heads*d_v]

Final shape: [bsz, seq_len, d_model]

PART5: Assembly of Transformer Block and Final Model

Structure of Transformer Block



- A RMSNorm module
- A MHA module
- Residual connection
- An other RMSNorm module
- A FFN module
- Residual connection

The parameters received by each Block are the union of all parameters required by the above modules!

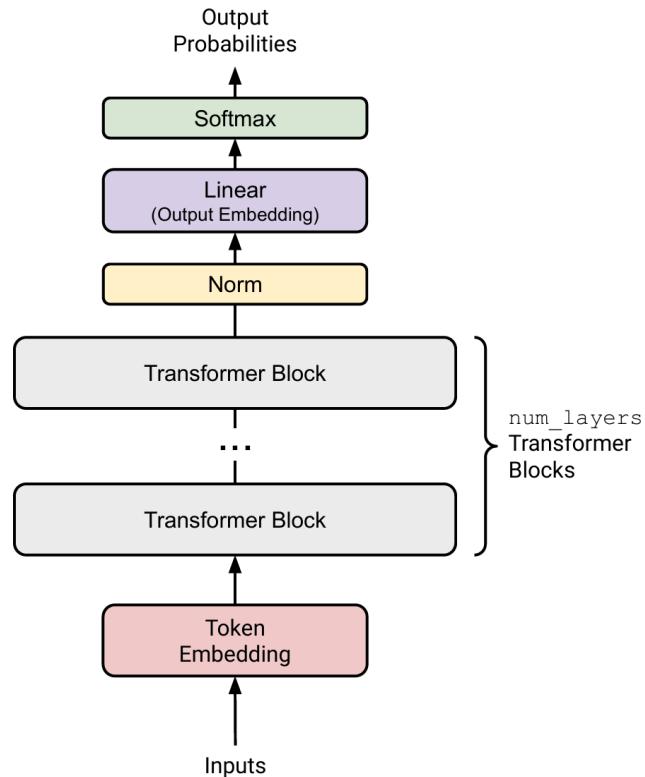
Assembly of Transformer Block

```
class Transformer_Block(nn.Module):
    def __init__(self,
                 d_model:int,
                 num_heads:int,
                 d_ff:int,
                 max_seq_length:int=None,
                 theta:int=None,
                 dtype=None,
                 device=None):
        super(Transformer_Block, self).__init__()
        self.RMSNorm_Attn=RMSNorm(d_model,dtype=dtype,device=device)
        self.RMSNorm_FF=RMSNorm(d_model,dtype=dtype,device=device)
        self.Multihead_Attn=Multihead_Attention(d_model,num_heads,max_seq_length,theta,device=device)
        self.Feed_Forward=Feed_Forward_Network(d_model,d_ff,device=device,dtype=dtype)
```

Assembly of Transformer Block

```
def forward(self,x:torch.Tensor,token_positions:torch.Tensor)->torch.Tensor:  
    residual_attn=x  
    x_normed_attn=self.RMSNorm_Attn(x)  
    attn_output=self.Multihead_Attn(x_normed_attn,token_positions)  
    x=residual_attn+attn_output  
  
    residual_ff=x  
    x_normed_ff=self.RMSNorm_FF(x)  
    ff_output=self.Feed_Forward(x_normed_ff)  
    x=residual_ff+ff_output  
  
    return x
```

Structure of Full Transformer



- A Token Embedding module
- Several Transformer Block modules
- A RMSNorm module
- Final Linear Transformation module

Assembly of Complete Transformer

```
class Transformer_LM(nn.Module):
    def __init__(self,
                 d_model:int,
                 num_heads:int,
                 d_ff:int,
                 vocab_size:int,
                 num_layers:int,
                 max_seq_length:int=None,
                 theta:int=None,
                 dtype=None,
                 device=None):
        super(Transformer_LM, self).__init__()
        self.d_model=d_model
        self.num_heads=num_heads
        self.d_ff=d_ff
        self.vocab_size=vocab_size
        self.num_layers=num_layers
        self.max_seq_length=max_seq_length
        self.theta=theta
        self.dtype=dtype
        self.device=device
        self.embeddings=Generate_EMBEDDINGS(vocab_size,
                                             d_model,
                                             device=device,
                                             dtype=dtype)
```

```
self.transformer_blocks=nn.ModuleList([
    Transformer_Block(d_model=d_model,
                      num_heads=num_heads,
                      d_ff=d_ff,
                      max_seq_length=max_seq_length,
                      theta=theta,
                      dtype=dtype,
                      device=device)
    for _ in range(num_layers)
])
self.final_norm=RMSNorm(d_model,device=device,dtype=dtype)
self.final_layer=Linear_Transform(d_model,
                                   vocab_size,
                                   device=device,
                                   dtype=dtype)
```

Assembly of Complete Transformer

```
def forward(self,token_ids:torch.Tensor,token_positions:torch.Tensor)->torch.Tensor:  
    x=self.embeddings(token_ids)  
    for block in self.transformer_blocks:  
        x=block(x,token_positions)  
    x=self.final_norm(x)  
    linear_score=self.final_layer(x)  
    return linear_score
```